

GENERATING THE FUNCTIONS WITH REGULAR GRAPHS UNDER COMPOSITION

THOMAS KERN

Department of Mathematics, Cornell University, Malott Hall, Ithaca, NY, 14853, USA
e-mail address: trk43@cornell.edu

ABSTRACT. While automata theory often concerns itself with regular predicates, relations corresponding to acceptance by a finite state automaton, in this article I study the regular functions, such relations which are also functions in the set-theoretic sense. Here I present a small (but necessarily infinite) collection of (multi-ary) functions which generate the regular functions under composition. To this end, this paper presents an interpretation of the powerset determinization construction in terms of compositions of input-to-run maps. Furthermore, known results using the Krohn-Rhodes theorem to further decompose my generating set are spelled out in detail, alongside some coding tricks for dealing with variable length words. This will include two clear proofs of the Krohn-Rhodes Theorem in modern notation.

1. INTRODUCTION

Automata theory is particularly fruitful in terms of equivalence theorems: regular expressions, deterministic and nondeterministic automata, the Myhill-Nerode theorem, the regular word logic and the weak second order theory of one successor all are equally expressive in the languages they describe. In this paper, I concern myself not with regular predicates (predicates which hold only for the words in a regular language) but with regular functions, functions whose behavior can be recognized by an automaton. This allows a translation of the Krohn-Rhodes theorem into yet another equivalent.

The Krohn-Rhodes Theorem concerns itself with *finite state transducers*, an abstraction of systems that:

- Accept inputs from a discrete set at discrete times,
- Retain some memory about previous inputs, which updates whenever an input is read,
- For each input read, produce some output from a discrete set based on the input and memory.

2012 ACM CCS: [Theory of computation]: Formal languages and automata theory—Transducers.

2010 Mathematics Subject Classification: Primary: 03D05; Secondary: 68Q45; 68Q70.

Key words and phrases: Krohn-Rhodes Theorem; Automata; Moore Machines.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grants No. DMS-0852811 and DMS-1161175. This paper is adapted from the first chapter of my dissertation research under Anil Nerode at Cornell University.

These are an abstraction of synchronous (as opposed to those that update continuously), digital (as opposed to those that deal with analog values) systems.

Originally proved in [7], the Krohn-Rhodes theorem itself gives a decomposition of arbitrary finite state transducers into a cascade of transducers from a small generating set. Computational implementations of this decomposition are available [3]. The Krohn-Rhodes Theorem can be used to analyze the rough behavior of automata, providing applications to Artificial Intelligence [4].

Finite state transducers are formalized as follows:

Definition 1.1. A *Moore Machine* is a tuple: $(\Sigma, Q, q_0, \Gamma, \delta, \epsilon)$.

- Σ is a finite set of input characters (alphabet).
- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- Γ is a finite set of output characters.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
- $\epsilon : \Sigma \rightarrow \Gamma$ is the output function.

For convenience, I will sometimes denote the map $x \mapsto \delta(a, x)$ by δ_a .

Given an input word $w \in \Sigma^*$, construct a run r and output o such that:

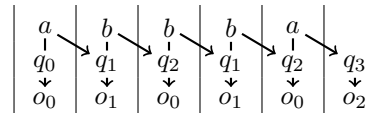
- $r[0] = q_0$.
- $r[i + 1] = \delta(r[i], w[i])$ for $0 \leq i < |w|$.
- $o[i] = \epsilon(r[i])$, for $0 \leq i \leq |w|$.

Where the indexing notation is such that:

$$w = w[0], \dots, w[|w| - 1].$$

The correct way of thinking about Moore Machines is that each input acts as a transition between one state and the next, or that each state is the state between inputs. A proper representation would have input characters half a step offset from states. Outputs are simply a product of the state the automaton is in and so should be in step with the states. However, representing the sequences of input characters and states as words requires a choice of direction to shift half a step. A considerable effort has been made to pick the option to result in the cleanest presentation. In this paper, the input character $w[i]$ tells the device how to transition from state $r[i]$ to state $r[i + 1]$ ¹.

Example 1.2. This example below shows how inputs, states, and outputs, respectively, line up according to my notation.



I will typically consider Moore Machines that simply output their states:

Definition 1.3. A Moore Machine is said to be *transparent* if $\Gamma = Q$ and ϵ is the identity. In this case, Moore Machines are presented as a tuple: (Σ, Q, q_0, δ) .

Moore Machines can be interpreted as functions from input words to output words:

¹This is as opposed to the input character $w[i]$ telling the device how to transition from state $r[i - 1]$ to state $r[i]$. This alternative is not uncommon.

Definition 1.4. Given a Moore Machine $M = (\Sigma, Q, q_0, \Gamma, \delta, \epsilon)$, and word $w \in \Sigma^*$, I denote by $M(w)$ the output of M on input w , with $M(w) \in \Gamma^*$.

Moore Machines, however, represent only a small subset of those functions on words which can be reasoned about using finite automata.

Definition 1.5. Given two alphabets, Σ, Γ , a function on words $f : \Sigma^* \rightarrow \Gamma^*$ is said to be:

- *length-preserving* if for any word w , $|f(w)| = |w|$.
- *causal* if $f(w)[i]$ depends only on $w[0], \dots, w[i]$.
- *strictly causal* if $f(w)[i]$ depends only on $w[0], \dots, w[i-1]$.
- *character-wise* if $f(w)[i]$ depends only on $w[i]$.

Definition 1.6. Define the following useful functions for dealing with words:

- Let S_a denote the *successor-a* function which appends the character a to the end of a word.
- Let Trunc denote the function which removes the last character of a word.
- Let Rest denote the function which removes the first character of a word.

Proposition 1.7. *Given a Moore Machine M , the function M computes is strictly causal, and increases length by 1.*

It is often more convenient to deal with length preserving versions of this function:

Definition 1.8. Given a Moore Machine $M = (\Sigma, Q, q_0, \Gamma, \delta, \epsilon)$, and word $w \in \Sigma^*$, denote by:

- $M^{\text{Trunc}}(w)$ the output of M on input w with the last state removed.
- $M^{\text{Rest}}(w)$ the output of M on input w with the first state (the start state) removed.

Proposition 1.9. *Given a Moore Machine M , the function M^{Trunc} is strictly causal and length preserving, and the function M^{Rest} is causal and length preserving.*

The notion of finite automaton, or finite state recognizer, is more commonly studied than the finite state transducer. *Finite state automata* are an abstraction of systems that:

- Accept inputs from a discrete set at discrete times,
- Retain some memory about previous inputs, which updates whenever an input is read,
- Having finished reading a sequence of inputs, either *accepts* or *rejects*.

Just as finite state transducers can be interpreted as functions on words, finite state automata can be interpreted as predicates on words, returning a boolean value after having read in a word. Their predicative nature means that finite state automata are more convenient to use in applications to formal logic. On the other hand, real world systems are more often interested in transforming inputs, and so are better represented by finite state transducers.

Finite automata yield a notion of a regular set or regular event, a collection of words or sequences of inputs which are exactly those which some finite automaton accepts. Once I define what it means to represent a function of words with an automaton, this will yield a notion of regular function.

Finite state recognizers are formalized as follows:

Definition 1.10. A *finite state automaton* is a tuple: $(\Sigma, Q, I, \delta, F)$.

- Σ is a finite set of input characters (alphabet).
- Q is a finite set of states.
- $I \subseteq Q$ is the set of initial states.

- $\delta : Q \times \Sigma \rightarrow Q$ is the transition relation.
- $F \subseteq Q$ is the set of final states.

Given an input word $w \in \Sigma^*$, $r \in Q^*$ is a run on input w if:

- $r[0] \in I$.
- $\delta(r[i], w[i], r[i+1])$ for $0 \leq i < |w|$.

w is *accepted* if there is a run r on input w such that $r[|w|] \in F$.

Definition 1.11. A finite state automaton $A = (\Sigma, Q, I, \delta, F)$ is *deterministic* if:

- I is a singleton.
- δ is a function from $Q \times \Sigma \rightarrow Q$, that is, given a $q \in Q$, and $a \in \Sigma$, there is a unique $q' \in Q$ such that $\delta(q, a, q')$.

By default, A is nondeterministic.

A well known theorem of finite automata is that:

Proposition 1.12. *If R is the set of accepted inputs of some automaton, then it is also the set of accepted inputs of some deterministic automaton.*

In either case, R is *regular*.

A common convention in logic is to identify a function f with the relation R_f which consists of all pairs of the form $(x, f(x))$, or, for n -ary functions,

$$(x_0, \dots, x_{n-1}, f(x_0, \dots, x_{n-1})),$$

for x in the domain of f . As such, a regular function can be defined as a relation which is regular and also a function. The question now is how to input multiple words, especially multiple words of different lengths, to a finite automaton ²

I introduce the Tuplefy map to merge words together in parallel so they can be read by an automaton. For words of different lengths, I add a dummy character $\#$.

Definition 1.13. Define the map

$$\text{Tuplefy} : \Sigma_0^* \times \dots \times \Sigma_{n-1}^* \rightarrow ((\Sigma_0 \cup \{\#\}) \times \dots \times (\Sigma_{n-1} \cup \{\#\}))^*,$$

Which satisfies:

$$\text{Tuplefy}(w_0, \dots, w_{n-1})[i]_j = \begin{cases} w_j[i] & \text{it exists} \\ \# & \text{otherwise} \end{cases}$$

and $|\text{Tuplefy}(w_0, \dots, w_{n-1})| = \max_i |w_i|$.

Example 1.14. Below is shown how Tuplefy combines words together into one word:

²Note that asynchronous input, that is, reading in the input words one at a time separated by a distinguished character is almost completely useless in terms of the functions that can be represented.

w_0	a	b	b	a	
w_1	a	b			
w_2					
w_3	b	b	b		
w_4	a	a	a	a	a
$\text{Tuplefy}(w_0, w_1, w_2, w_3, w_4)$	$\begin{pmatrix} a \\ a \\ \# \\ b \\ a \end{pmatrix}$	$\begin{pmatrix} b \\ b \\ \# \\ b \\ a \end{pmatrix}$	$\begin{pmatrix} b \\ \# \\ \# \\ b \\ a \end{pmatrix}$	$\begin{pmatrix} a \\ \# \\ \# \\ \# \\ a \end{pmatrix}$	$\begin{pmatrix} \# \\ \# \\ \# \\ \# \\ a \end{pmatrix}$

Now I can define the notion of regular relation and regular function:

Definition 1.15. An n -ary relation $R \subseteq \Sigma_0^* \times \cdots \times \Sigma_{n-1}^*$ is *regular* if there is a finite automaton A with input alphabet $(\Sigma_0 \cup \{\#\}) \times \cdots \times (\Sigma_{n-1} \cup \{\#\})$ such that:

$$R(w_0, \dots, w_{n-1}) \iff A \text{ accepts } \text{Tuplefy}(w_0, \dots, w_{n-1}).$$

An n -ary function $f : \Sigma_0^* \times \cdots \times \Sigma_{n-1}^* \rightarrow \Sigma_n^*$ is *regular* if there is a finite automaton A with input alphabet $(\Sigma_0 \cup \{\#\}) \times \cdots \times (\Sigma_n \cup \{\#\})$ such that:

$$f(w_0, \dots, w_{n-1}) = w_n \iff A \text{ accepts } \text{Tuplefy}(w_0, \dots, w_n).$$

Regular relations and functions are a key part of the analysis of various automaton logics. For instance:

Definition 1.16. Given a finite alphabet Σ , let $\mathcal{W}_\Sigma = (\Sigma^*, \leq, =_{el}, S_a|_{a \in \Sigma})$ where:

- \leq is the prefix relation on words,
- $=_{el}$ is the equal length relation on words,
- S_a is the *Successor- a* unary operation, which appends an a onto the end of a word.

I call \mathcal{W}_Σ the *regular word logic over Σ* ³.

Proposition 1.17. If R is a relation on Σ^* , the following are equivalent:

- R is regular,
- R is given by a formula ϕ in the language of \mathcal{W}_Σ .

Hence my interest in regular functions. If $\phi(x_0, \dots, x_n)$ is a formula in the language of \mathcal{W}_Σ such that:

$$\forall x_0, \dots, x_{n-1} \exists x_n : \phi(x_0, \dots, x_n),$$

Then, since lexicographic ordering $<_L$ is regular and well-founded, the following relation is regular

$$\psi(x_0, \dots, x_n) \iff \phi(x_0, \dots, x_n) \wedge \nexists y : [y <_L x_n \wedge \phi(x_0, \dots, y)],$$

And also a function. Restrictions of this sort are called *Skolem functions*.

It is also worth noting that a classification of the regular functions also yields a classification of the regular languages, since for any regular language R , the characteristic function

$$\mathcal{X}_R : w \mapsto \begin{cases} 1 & w \in R \\ 0 & w \notin R \end{cases}$$

³While this language may at first seem artificial, it is equally expressive with the *Weak Second Order Theory of One Successor*, the theory of natural numbers, finite sets of natural numbers, the +1 operation, and containment.

Is regular.

The idea of achieving quantifier elimination on the regular word logic via function composition was inspired in part by a theorem in [1], which decomposes regular predicates in terms of shifting operations, character-wise operations, and a univocal regular predicate.

Finally, in this section, I connect Moore Machines and Finite Automata:

Proposition 1.18. *Given a Moore Machine $M = (\Sigma, Q, q_0, \Gamma, \delta, \epsilon)$, the functions M , M^{Trunc} , and M^{Rest} are regular.*

Proof. Construct a deterministic finite automaton that keeps track of two pieces of information: the state the Moore Machine is expected to be in at any particular point, and a boolean value to keep track of whether the proposed output has so far been correct. Additionally, in the case of M , there will be a final character of the form $\begin{pmatrix} \# \\ o \end{pmatrix}$ and a small amount of information must be kept track of to handle this correctly.

For example, for M^{Trunc} , let:

$$A = (\Sigma \times \Gamma, Q \times \{0, 1\}, (q_0, 0), \delta', Q \times \{0\}),$$

Where:

$$\delta'((q, i), (a, o)) = \left(\delta(q, a), \begin{cases} 0 & i = 0 \wedge \epsilon(q) = o \\ 1 & \text{otherwise} \end{cases} \right).$$

□

Proposition 1.19. *Every strictly causal, length-preserving, regular function is given by M^{Trunc} for some Moore Machine M . Every causal, length-preserving function is given by M^{Rest} for some Moore Machine M .*

Proof. I prove this for a binary, strictly causal, length-preserving, regular function f . The proof is nearly identical in the general case. Let $f : \Sigma \rightarrow \Gamma$ be given. By Proposition 1.17, there is a formula $\phi(w_0, w_1)$ in the language $\mathcal{W}_{\Sigma \cup \Gamma}$ such that

$$\phi(w_0, w_1) \iff f(w_0) = w_1.$$

Since f is a strictly causal function, the first $n - 1$ characters of the input determine the n th character of the output. By simple tricks in $\mathcal{W}_{\Sigma \cup \Gamma}$, one can construct a formula ϕ_u for each $u \in \Gamma$ such that ϕ_u is true of exactly those sequences of characters that produce an output of u in the next place. These ϕ_u describe a collection of regular sets which partition all of Σ^* . Let A_u be a finite automaton that recognizes the corresponding collection.

Now to construct the Moore Machine M . It should run each of the A_u in parallel to determine its state. Since the A_u recognize disjoint collections, exactly one of the A_u will be in an accept state at any time. The ϵ function for the Moore Machine should take in the tuple of states for the A_u and output the one which is in an accept state. It suffices now to check that M^{Trunc} is identically f .

The proof for M^{Rest} is similar.

□

Of course, there are plenty of other regular functions. In this paper I provide a small set of functions whose closure under multi-ary composition generates all of them. In the next section, I will interpret the proof of Proposition 1.12 to reduce the problem to studying functions given by the actions of Moore Machines and Reverse Moore Machines. Moore Machines allow one to construct functions which transmit information only to the right (towards the end of the inputs), but they need to be combined with a method to transmit

information to the left. Consider, for instance, the Rest function, which removes the first character of a string, shifting to the left. Alternately, a function which outputs 00 or 01 depending on whether there are an even or odd number of 0s in the input. These functions are regular, but not causal.

2. DETERMINIZATION AND HARVESTING

In this section, I define Reverse Moore Machines, and provide a technique for decomposing a length-preserving regular function as a multivariable composition of a Moore Machine function, a Reverse Moore Machine function, and character-wise maps to connect them. This will require several stages. First, I will briefly discuss some notation for discussing character-wise functions. Second, I will introduce the notion of a Reverse Moore Machine similar to the Moore Machines introduced in section 1. Third, I will present the decomposition. This decomposition is based on the classical powerset determinization construction, viewed from a novel perspective. In the next section, I will discuss handling general regular functions.

First, some notation for functions which operate character-wise:

Definition 2.1. Given two finite alphabets Σ, Γ , and a function $f : \Sigma \rightarrow \Gamma$, call the function $\text{Cw}_f : \Sigma^* \rightarrow \Gamma^*$ which applies f to each character of the input, the *character-wise f map*.

If f is an n -ary function for $n > 1$, one can also make sense of Cw_f . Let

$$f : \Sigma_0 \times \cdots \times \Sigma_{n-1} \rightarrow \Gamma.$$

Then one defines the partial function

$$\text{Cw}_f : \Sigma_0^* \times \cdots \times \Sigma_{n-1}^* \rightarrow \Gamma^*,$$

which takes in inputs of all the same lengths and produces an output of the same length, where:

$$\text{Cw}_f(w_0, \dots, w_{n-1}) = \text{Cw}_f(\text{Tuplefy}(w_0, \dots, w_{n-1})).$$

Noting that *Tuplefy* does not produce characters with $\#$ in them for equal-length inputs.

Of course, Cw_f is causal and length preserving (and reverse-causal, when I define the notion). Every Moore Machine function can be written as the action of the corresponding transparent Moore Machine composed with a bitwise application of its ϵ function.

I now define some notation for dealing with Reverse Moore Machines, analogous to the notation established previously.

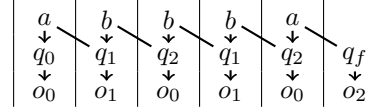
Definition 2.2. A *Reverse Moore Machine* is a tuple: $(\Sigma, Q, q_f, \Gamma, \delta, \epsilon)$.

- Σ is a finite set of input characters (alphabet).
- Q is a finite set of states.
- $q_f \in Q$ is the final state.
- Γ is a finite set of output characters.
- $\delta : Q \times \Sigma \rightarrow Q$ is the reverse transition function.
- $\epsilon : \Sigma \rightarrow \Gamma$ is the output function.

Given an input word $w \in \Sigma^*$, construct a run r and output o such that:

- $r[|w|] = q_f$.
- $r[i] = \delta(r[i+1], w[i])$ for $0 \leq i < |w|$.
- $o[i] = \epsilon(r[i])$, for $0 \leq i \leq |w|$.

Example 2.3. Here I show below how inputs, states, and outputs, respectively, line up according to this notation.



To prevent type mismatches, I will denote Reverse Moore Machines with letters R, P as opposed to letters M, N for Moore Machines.

Definition 2.4. A Reverse Moore Machine is said to be *transparent* if $\Gamma = Q$ and ϵ is the identity. In this case, the Reverse Moore Machine is presented as a tuple: (Σ, Q, q_f, δ) .

As before, one can interpret Reverse Moore Machines as functions from input words to output words:

Definition 2.5. Given a Reverse Moore Machine $R = (\Sigma, Q, q_f, \Gamma, \delta, \epsilon)$, and word $w \in \Sigma^*$, denote by:

- $R(w)$ the output of R on input w .
- $R^{\text{Trunc}}(w)$ the output of R on input w with the last state removed.
- $R^{\text{Rest}}(w)$ the output of R on input w with the first state (the start state) removed.

Analogous to the notions of causal and strictly causal, one has notions of reverse causal and strictly reverse causal:

Definition 2.6. Given two alphabets, Σ, Γ , a function on words $f : \Sigma^* \rightarrow \Gamma^*$ is said to be:

- *reverse causal* if $f(w)[i]$ depends only on $w[i], w[i+1], \dots$
- *strictly reverse causal* if $f(w)[i]$ depends only on $w[i+1], w[i+2], \dots$

Of course, a function is character-wise iff it is causal and reverse causal.

One also has a notion of reverse deterministic automaton:

Definition 2.7. A finite automaton $A = (\Sigma, Q, I, \delta, F)$ is *reverse deterministic* if:

- F is a singleton.
- For each q, a , there is a unique q' such that $(q', a, q) \in \delta$.

A reverse deterministic automaton can also be viewed as a transparent Reverse Moore Machine.

I now have the notation to state the main result:

Theorem 2.8. *Any length-preserving regular function can be written as a multivariable composition of the form:*

$$f(w) = R^{\text{Trunc}}(\text{Cw}_{\text{Pair}}(w, M^{\text{Trunc}}(w))).$$

It's worth noting that Cw_{Pair} has the same action here as Tuplefy. I've chosen to use Cw_{Pair} here to indicate that I'm not using the length-padding features of Tuplefy.

A few lemmas must be proved first:

Lemma 2.9. *Given a length-preserving regular function $f : \Sigma^* \rightarrow \Gamma$, there is an automaton B with state set Q and map $\epsilon : Q \rightarrow \Gamma$ such that $f(w)$ is $\text{Trunc} \circ \text{Cw}_\epsilon$ applied to any run of B on input w .*

Proof. Given the length-preserving function $f : \Sigma^* \rightarrow \Gamma^*$, let:

$$C = ((\Sigma \cup \{\#\}) \times (\Gamma \cup \{\#\}), Q, I, \delta, F)$$

Witness the regularity of f . Since C automatically rejects any inputs with a $\#$ in them, one can restrict it to:

$$A = (\Sigma \times \Gamma, Q, I, \delta, F)$$

Recognizing the same set of inputs.

Construct an automaton:

$$B = (\Sigma, Q \times \Gamma, I \times \Gamma, \delta', F \times \Gamma),$$

Where:

$$((q, o), a, (q', o')) \in \delta' \iff (q, (a, o), q') \in \delta$$

B is nondeterministic. I assert that runs of B on input w will necessarily have Γ component $S_o(f(w))$, for some $o \in \Gamma$. Firstly, it should be clear that if r is an accepting run of A on input $\text{Tuplefy}(w, f(w))$, then $\text{Tuplefy}(r, S_o(f(w)))$ is an accepting run of B on input w for any $o \in \Gamma$.

Now, suppose $\text{Tuplefy}(r', g)$ is an accepting run of B on input w . Then it is easy to check that r' is an accepting run of A on input $\text{Tuplefy}(w, \text{Trunc}(g))$. Since A witnessed f being a regular function, $\text{Trunc}(g)$ must be $f(w)$. This completes the proof for ϵ taking the Γ component of the states of B . \square

Definition 2.10. Given a nondeterministic automaton $A = (\Sigma, Q, I, \delta, F)$, define its *determinization*:

$$\det(A) = (\Sigma, \mathcal{P}(Q), \{I\}, \delta', \{E \subset Q : E \cap F \neq \emptyset\}),$$

Where:

$$\delta'(K, a) = \{q' \in Q \mid \exists q \in K : \delta(q, a, q')\}.$$

Typically throughout this paper I will be concerned with the determinization as a transparent Moore Machine, so the set of final states doesn't matter much.

Definition 2.11. Given a nondeterministic automaton $A = (\Sigma, Q, I, \delta, F)$, define its *harvester*:

$$\text{harv}(A) = (\Sigma \times \mathcal{P}(Q), Q \cup \{q_F\}, Q \cup \{q_F\}, \delta', \{q_F\}),$$

Where:

- The q such that $\delta'(q, (a, K), q')$ (for $q' \in Q$) is given by the least $q'' \in K$ such that $\delta(q'', a, q')$ or q_F if none exist.
- The q such that $\delta'(q, (a, K), q_F)$ is given by finding the least $q' \in F$ such that there is a $q'' \in K$ such that $\delta(q'', a, q')$ and then having q be the least $q'' \in K$ such that $\delta(q'', a, q')$. If no such $q' \in F$ exists, then q is given by q_F .

The powerset determinization of an automaton A produces an automaton that keeps track of, at every position, the set of states of A which are reachable through some sequence of transitions, having read the input up to that point. However, not every one of these reachable states necessarily shows up in some accepting run: it may be that being in one state now means later on having to be in another state which it cannot transition out of, or that being in a state now dooms the automaton to being in a reject state once it has finished reading the input. In order to use the determinization to find an accepting run of the original automaton, provided there is an accepting run, one needs to start at the

end and work backwards, all the while staying within states that one knows can be traced through a sequence of transitions back to a start state at the beginning.

Specifically, if the automaton is in a state q which is reachable through some sequence of transitions after having read $S_a(w)$, then there must be at least one state q' which is reachable through some sequence of transitions after having read w such that reading a takes it from state q' to state q .

It is necessary to introduce an additional dummy state q_F to start out in to make sure the harvester automaton is reverse-deterministic. Although this application of the harvester automaton will see only pairs of the form (a, K) for a some symbol being read by the original automaton and K the set of states reachable immediately prior to reading that specific a , the automaton should be prepared to read in arbitrary input pairs. An invalid input will cause the reverse deterministic automaton to go into the dummy q_F state. Additionally, a cheap fix is necessary to account for the fact that one doesn't know what state the run of the original automaton ends on and one needs a single "final" state for the reverse deterministic automaton to "start out" in. The dummy q_F state represents all final states which are reachable. Truncating the run will remove this dummy state.

Lemma 2.12. *Suppose one has a nondeterministic automaton A , and valid input w . Let p be the run of $\det(A)$ on input w . I claim that $\text{harv}(A)$ on input $\text{Tuplefy}(w, \text{Trunc}(p))$ will have run r , an accepting run of A on input w .*

Proof. It suffices to show that the only occurrence of q_F in r is as the final character. By construction, $\text{harv}(A)$ will satisfy the transition relations. As mentioned before the construction of $\text{harv}(A)$ also prevents backwards transitioning into the q_F state for this particular input, since a reachable state can always be traced back to a reachable state. \square

By lemma 2.9, given a regular, length-preserving function f , there is a nondeterministic automaton B which takes in a word w and has a run r which projects to $f(w)$. By Lemma 2.12:

$$r = \text{harv}(B)^{\text{Trunc}}(\text{CwPair}(w, \det(B)^{\text{Trunc}}))$$

By modifying the outer Reverse Moore Machine, one can throw in the appropriate projection to its output map to produce $f(w)$. This completes the proof of Theorem 2.8.

This is a remarkable result. Every length-preserving regular function can be computed in a two-step process: one pass forwards through the input leaving behind some information, then a pass backwards through the input with this additional information to directly produce the output. Two passes suffice; having more passes doesn't increase the expressive power.

3. LENGTH MODIFICATION

The only functions this paper has dealt with so far were length-preserving. In this section, I show that most of the interesting behavior of regular functions was already captured in the length-preserving case.

Proposition 3.1. *Suppose $f : \Sigma \rightarrow \Gamma$ is a regular function. Then there is a fixed constant c associated to f such that $f(w)$ is no longer than $c + |w|$ for every w .*

The proof is based on the pumping lemma.

Proof. Let A be a deterministic automaton with c states accepting exactly words of the form $\text{Tuplefy}(w, f(w))$. Choose a specific w and suppose $f(w)$ is longer than $c + |w|$. Imagine

what happens as A reads in $\text{Tuplefy}(w, f(w))$, specifically, after w is finished, and A is reading in characters of the form $\begin{pmatrix} \# \\ o \end{pmatrix}$ for some $o \in \Gamma$. Because there are more positions like this than there are states of A , by the pigeonhole principle some two positions will have the same state, say at positions i and j . Note however that if one removes all positions in $\text{Tuplefy}(w, f(w))$ between i and j (including i , excluding j) one still has an accepting run, of the form $\text{Tuplefy}(w, g)$ for some g strictly shorter than $f(w)$. This contradicts the assumption that A accepted exactly words of the form $\text{Tuplefy}(w, f(w))$. \square

From this, one can also show that for any n -ary regular function f , there is a fixed constant c associated to f such that $f(w_0, \dots, w_{n-1})$ is no longer than c plus the length of the maximum input.

Note that the automaton A in the proof never encounters the input character $\begin{pmatrix} \# \\ \# \end{pmatrix}$. As such, one may assume that this character acts as the identity on the states of A . The resulting automaton recognizes all pairs of the form $\text{Tuplefy}(S_{\#}^b(w), f(w))$ for arbitrary b ($S_{\#}^c$ simply represents a c -fold composition of the $S_{\#}$ function). By adding in a counter, one can recognize exactly the pairs of the form $\text{Tuplefy}(S_{\#}^c(w), f(w))$.

As such, the function which takes $S_{\#}^c(w)$ to $S_{\#}^d(f(w))$ for $d = |f(w)| - |w|$ is a length-preserving regular function g , and thus can be written as a composition of character-wise maps, truncated Moore Machines, and truncated Reverse Moore Machines as in Theorem 1. Now f can be written as:

$$f(w) = \text{Unpad}(g(S_{\#}^c(w))),$$

Where Unpad removes final $\#$ characters. Note that one does not know how many final $\#$ characters there will be. For functions which reduce length, the number will be more than c and could be as much as $c + |w|$. One might be tempted to try to replace Unpad with some function like $S_{\#}^{-1}$ (or, even less suited to the task, Trunc), which either removes a single final $\#$ or leaves the word alone if it cannot. However, since there are regular functions which take words of arbitrary length and reduce them to length 1, the generating set needs a generator that can produce unbounded shortening as well.

Note that if f is n -ary for $n > 1$, it follows that:

$$f(w_0, \dots, w_{n-1}) = \text{Unpad}(g(\text{Tuplefy}(S_{\#}^c(w_0), \dots, S_{\#}^c(w_{n-1})))),$$

For some regular, length-preserving function g .

As such:

Theorem 3.2. *Any regular function can be written as a multivariable composition of:*

- *Truncated Moore Machines,*
- *Truncated Reverse Moore Machines,*
- *Character-wise maps,*
- *Tuplefy (allowing one to generate multiary character-wise maps),*
- S_a for various a ,
- Unpad .

It's worth noting here that this generating set is infinite. Specifically, there are an infinite number of Moore Machines and Reverse Moore Machines. There are also an infinite number of Character-wise maps, but this isn't essential – one could use encoding methods to work purely with a single two-character alphabet (plus, optionally, the dummy character $\#$).

The infinitude of the generating set, however, is essential. The easiest way to see this is to talk about period introduction. Provided the input to a regular function has a sufficiently long periodic portion in the middle, the output of the regular function will also have a long periodic portion in the middle (it suffices to verify this of the generators above). What's more, the period of the periodic portion of the output can only have prime factors which show up either in the periods of the periodic portions of the inputs or which are smaller than the number of states of the associated automaton to the regular function. However, one can easily build regular functions which introduce any prime factor into the periodicity of their inputs, so there must not be any bound on the sizes of associated automata to regular functions in the generating set. In summary:

Proposition 3.3. *Any set of regular functions which generates all regular functions under multivariable composition must be infinite.*

I conclude this paper with a discussion of known results regarding the Krohn-Rhodes Theorem. First, I will provide a proof of the Krohn-Rhodes Theorem adapted from Ginzburg [6]. Then I will use the Krohn-Rhodes Theorem to decompose the Moore Machine and Reverse Moore Machine generators into smaller, simpler generators. The idea of interpreting a the cascade given by the Krohn-Rhodes theorem as a composition of functions can be found in [5]. This paper will spell out this composition precisely and in modern notation, as has been done for previous compositions. A few sections will be dedicated to cleaning up the resulting set of generators, followed by proposed future research.

4. THE KROHN-RHODES THEOREM

In this section, I present the Krohn-Rhodes Theorem as adapted to the context of multivariate composition of regular functions. The original proof of the Krohn-Rhodes Theorem, in [7], was presented in terms of wreath products of semigroups. More modern presentations of the Krohn-Rhodes Theorem typically present it in terms of the cascade product of finite state transducers.

The cascade product of two transducers M_1, M_2 is a single system consisting of both machines. First, machine M_2 reads in both the input to the system and the current state of M_1 to update its state. When it has finished, machine M_1 updates its state based only on the input to the system. Finally, an output is produced based on the states of M_1 and M_2 . This reflects the reality of systems where updating the states of machines takes a small but appreciable amount of time. In a well designed system, M_2 should not have to wait for M_1 to finish its update before it can update its state. As such, M_2 uses the state of M_1 prior to reading the input to update.

The Krohn-Rhodes Theorem separates out two extremes of behavior for finite automata. In general, reading in an input character induces a function on the states of the automaton. This function may map two states to the same state or to separate states. At one extreme, it may act as a *permutation* in which case it maps all states to separate states. In this case, it is possible to undo this action. One can recover the state before reading a character which acts as a permutation, provided one knows which character the automaton read. At the other extreme, an input character may act as a *reset* in which case it maps all states to the same state. In this case all information about the previous state is lost.

Definition 4.1. A Moore Machine or deterministic automaton is said to be:

- A *permutation automaton* if each of its inputs acts as a permutation on its states,

- A *reset automaton* if each of its inputs acts as a reset or the identity on its states,
- A *permutation-reset automaton* if each of its inputs acts as a permutation or a reset on its states.

I now state the Krohn-Rhodes theorem, in a bit of an unusual fashion:

Theorem 4.2 (Krohn-Rhodes). *Given a transparent Moore Machine M , one can write its truncated action M^{Trunc} as a multivariable composition of truncated actions of permutation-reset Moore Machines M_0, \dots, M_{n-1} for n the number of states of M , and a final character-wise map f . What's more, this composition takes on a fairly simple form. Let:*

$$\begin{aligned} w_0 &= M_0^{\text{Trunc}}(w), \\ w_1 &= M_1^{\text{Trunc}}(\text{Tuplefy}(w, w_0)), \\ w_2 &= M_2^{\text{Trunc}}(\text{Tuplefy}(w, w_0, w_1)), \\ &\vdots \\ w_{n-1} &= M_{n-1}^{\text{Trunc}}(\text{Tuplefy}(w, w_0, \dots, w_{n-2})), \end{aligned}$$

Then:

$$M^{\text{Trunc}}(w) = \text{Cw}_f(w_0, \dots, w_{n-1}).$$

Every map in this composition is length-preserving. Of course one can write this as simply one large multivariable composition of a character-wise map and truncated actions of permutation-reset transparent Moore Machines, but this is unwieldy to write down. The above also presents an efficient way of computing the composition, although readers concerned with efficiency are encouraged to look into the Holonomy decomposition [3].

The proof is inductive: I show that for every transparent Moore Machine M , there's another transparent Moore Machine \bar{M} that keeps track of a state M is not in in a permutation-reset way. This reduces the amount of information that needs to be kept track of by one state, and one can keep doing this until one has kept track of all the information to know what state M is in.

The proof in [6] allows for the possibility that one can keep track of several states the automaton M is not in in a permutation-reset way at the same time, as opposed to one at a time in the proof below. This is more efficient, but adds needless complexity to the proof.

The key piece of the construction is the Permutation-Reset Lemma, below.

Lemma 4.3 (Permutation-Reset Lemma). *Given two finite ordered sets of the same size I, J , and map between them f , there is a map $g : I \rightarrow J$ such that:*

- *g either acts as:*
 - *A bijection from I to J (a permutation on the position indices),*
 - *Or has singleton image (a reset on position indices),*
- *And for $x, y \in I$, with $x \neq y, f(x) \neq g(y)$.*
- *For any $x \in I$, f maps elements of $I \setminus \{x\}$ to $J \setminus \{g(x)\}$.*

Proof. Suppose f does act as a bijection. Then $g = f$ is a permutation and satisfies the inequality condition.

Suppose f does not act as a bijection. Then g which maps everything to the smallest element of J which is not in the image of f has singleton image and satisfies the inequality condition.

The third condition is just a rephrasing of the second, but will come in handy later on. \square

A specific application of the Permutation-Reset Lemma is that one can have a transparent Moore Machine that keeps track of a state the original transparent Moore Machine is not in:

Lemma 4.4. *Given a transparent Moore Machine $M = (\Sigma, Q, q_0, \delta)$, there is a permutation-reset transparent Moore Machine \overline{M} with the same state set such that on input w , the state of M at any one time is not the state of \overline{M} .*

Proof. Assign a natural ordering to Q . Define

$$\overline{M} = (\Sigma, Q, \overline{q}_0, \overline{\delta}),$$

Where \overline{q}_0 is the smallest element in Q which is not q_0 , and $\overline{\delta}(q, a)$ is given by:

- $\delta(q, a)$ if a acts as a permutation.
- Otherwise, the smallest $q' \in Q$ which is not in the image of any state under the action of a .

If the action of a was a permutation originally, it is still a permutation in the new automaton. This permutation not only maps the state the automaton is in before reading a to the state afterwards, but also from a state the automaton is not in before reading a to a state the automaton is not in afterwards. In the second case, note that the choice of q' does not depend on q , so this action is a reset. Obviously, it maps a state the original automaton is not in before reading a to a state the automaton is not in after reading a . \square

Lemma 4.5. *Given transparent Moore Machines M, \overline{M} as above with state sets Q , there is a third transparent Moore Machine:*

$$\widehat{M} = (\Sigma \times Q, \{0, \dots, |Q| - 2\}, \hat{i}_0, \widehat{\delta}),$$

Such that for any input w , if M on reading w winds up in state q , and \overline{M} on reading w winds up in state \overline{q} then \widehat{M} on reading $\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))$ will wind up in state \hat{i} , where q is the element in position⁴ \hat{i} of $Q \setminus \{\overline{q}\}$.

Proof. Let $\widehat{M} = (\Sigma \times Q, \{0, \dots, |Q| - 2\}, \hat{i}_0, \widehat{\delta})$ where \hat{i}_0 is the index of q_0 in $Q \setminus \{\overline{q}_0\}$ and $\widehat{\delta}(i, (a, \overline{q}))$ is computed by:

- Computing $q' = \overline{\delta}(\overline{q}, a)$. This is the state \overline{M} says that M is not in after reading a .
- Computing q , the i th element of $Q \setminus \{\overline{q}\}$. This is the state of M prior to reading a .
- Return j , the index of $\delta(q, a)$ in $Q \setminus q'$.

The above construction is designed specifically to satisfy the conclusion. \square

⁴To maintain notational consistency within this paper, where ordered collections are indexed starting with 0, I refer to the first element of a set as being in position 0, and generally the $i + 1$ st element of a set as being in position i . To avoid confusion, I avoid using the notation “ i th element”, instead using notation “the element in position i ”.

As such:

$$M^{\text{Trunc}}(w) = \text{Cw}_p(\overline{M}^{\text{Trunc}}(w), \widehat{M}^{\text{Trunc}}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))))$$

Where $p(\bar{q}, i)$ is the i th element of $Q \setminus \{\bar{q}\}$.

Finally, I prove the Krohn-Rhodes Theorem:

Proof. By induction on the number of states of M .

Base Case: If M has one state, then f in the composition is 0-ary, and one can have it just output that constant state.

Inductive Case: Given M , one can write:

$$M^{\text{Trunc}}(w) = \text{Cw}_p(\overline{M}^{\text{Trunc}}(w), \widehat{M}^{\text{Trunc}}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))))$$

With \overline{M} permutation-reset. By the inductive hypothesis, one can write $\widehat{M}^{\text{Trunc}}(w)$, which has one fewer state than M , as:

$$\widehat{M}^{\text{Trunc}}(w) = \text{Cw}_f(w_1, \dots, w_{n-1}),$$

Where:

$$\begin{aligned} w_1 &= M_1^{\text{Trunc}}(w) \\ w_2 &= M_2^{\text{Trunc}}(\text{Tuplefy}(w, w_1)) \\ w_3 &= M_3^{\text{Trunc}}(\text{Tuplefy}(w, w_1, w_2)) \\ &\vdots \\ w_{n-1} &= M_{n-1}^{\text{Trunc}}(\text{Tuplefy}(w, w_1, \dots, w_{n-2})) \end{aligned}$$

As such, $\widehat{M}^{\text{Trunc}}(\text{Tuplefy}(w, \widehat{M}^{\text{Trunc}}(w)))$ is given by just plugging in:

$$\widehat{M}^{\text{Trunc}}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))) = \text{Cw}_f(w_1, \dots, w_{n-1}),$$

Where:

$$\begin{aligned} w_1 &= M_1^{\text{Trunc}}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))), \\ w_2 &= M_2^{\text{Trunc}}(\text{Tuplefy}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w)), w_1)), \\ w_3 &= M_3^{\text{Trunc}}(\text{Tuplefy}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w)), w_1, w_2)), \\ &\vdots \\ w_{n-1} &= M_{n-1}^{\text{Trunc}}(\text{Tuplefy}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w)), w_1, \dots, w_{n-2})). \end{aligned}$$

Alternately, fiddling with some parentheses in the definitions in the automata:

$$\widehat{M}^{\text{Trunc}}(\text{Tuplefy}(w, \overline{M}^{\text{Trunc}}(w))) = \text{Cw}_f(w_1, \dots, w_{n-1},)$$

Where:

$$\begin{aligned}
w_0 &= \overline{M}^{\text{Trunc}}(w), \\
w_1 &= M_1^{\text{Trunc}}(\text{Tuplefy}(w, w_0)), \\
w_2 &= M_2^{\text{Trunc}}(\text{Tuplefy}(w, w_0, w_1)), \\
w_3 &= M_3^{\text{Trunc}}(\text{Tuplefy}(w, w_0, w_1, w_2)), \\
&\vdots \\
w_{n-1} &= M_{n-1}^{\text{Trunc}}(\text{Tuplefy}(w, w_0, \dots, w_{n-2})).
\end{aligned}$$

In which case:

$$M^{\text{Trunc}}(w) = \text{Cw}_p(\underbrace{\overline{M}^{\text{Trunc}}(w)}_{w_0}, \text{Cw}_f(w_1, \dots, w_{n-1})).$$

One can combine p and f to get a single character-wise function on w_0, \dots, w_{n-1} , thus completing the induction. \square

The proof is still straightforward if one unwinds the induction. In the construction, w_0 is keeping track of a state M is not in, but it may as well be keeping track of an index for a state M is not in. w_1 is keeping track of an index of a state M is not in once one has removed the state in position w_0 from Q . w_2 is keeping track of an index of a state M is not in once one has removed the states that w_1 and w_2 are keeping track of from Q . And so forth. One can formalize this indexed removal process as follows:

Definition 4.6. Given a positive integer n , an *ordinal removal sequence* for n is a (possibly empty) sequence of positive integers (k_0, \dots, k_i) satisfying:

$$\begin{aligned}
i &< n, \\
0 &\leq k_j < n - j.
\end{aligned}$$

One can interpret an ordinal removal sequence for n as a series of commands operating on an ordered set of size n of the form “remove the $i + 1$ st smallest element remaining.” Note that as elements are removed, there are fewer elements remaining, hence the decreasing upper limit on k_j in the second constraint. Consistent with the rest of this paper, I begin indexing with 0, so a 0 means remove the smallest element.

Definition 4.7. Given an ordered set L , and an ordinal removal sequence for $|L|, \mathbf{k} = (k_0, \dots, k_{i-1})$, define $\text{Remove}(L, \mathbf{k})$ recursively:

- $\text{Remove}(L, ()) = L$,
- $\text{Remove}(L, (k_0, \dots, k_{i-1}))$ is given removing the element in position k_i (with order inherited from L , and 0 means remove the smallest element) of $\text{Remove}(L, (k_0, \dots, k_{i-2}))$.

Let $\mathcal{O}_{k,n}$ denote the set of ordinal removal sequences for n of length k .

Recall that \frown is used for concatenation, so:

$$(k_0, \dots, k_{i-1}) \frown j = (k_1, \dots, k_{i-1}, j).$$

Example 4.8. Consider the ordinal removal sequence for 5: $(0, 1, 2, 1)$ acting on the ordered set (A, B, C, D, E) :

Start:	(A, B, C, D, E)
Remove at position 0:	(B, C, D, E)
Remove at position 1:	(B, D, E)
Remove at position 2:	(B, D)
Remove at position 1:	(B)

The following lemmat formally gives the construction of the M_j in the Krohn-Rhodes decomposition.

Lemma 4.9. *Given a transparent Moore Machine $M = (\Sigma, Q, q_0, \delta)$, and $0 \leq j < |Q| - 1$, there is a permutation-reset transparent Moore Machine*

$$M_j = (\Sigma \times \mathcal{O}_{j,|Q|}, \{0, \dots, |Q| - j\}, k_0, \delta'),$$

Such that if κ is a word of ordinal removal sequences for $|Q|$ of length j , that is, $\kappa \in (\mathcal{O}_{j,|Q|})^$, satisfying:*

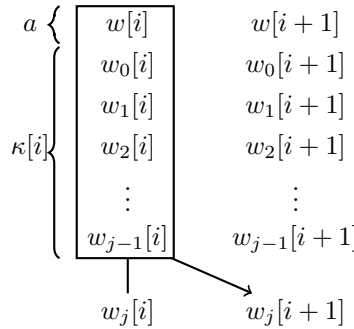
- *At any point i , $\kappa[i]$ does not remove $M(w)[i]$ from Q . That is, $M(w)[i] \in \text{Remove}(Q, \kappa[i])$.*
- *$\kappa[i + 1]$ is determined by $\kappa[i]$ and $w[i]$, specifically such that:*
- *The map δ_a (from M) maps states in $\text{Remove}(Q, \kappa[i])$ to states in $\text{Remove}(Q, \kappa[i + 1])$.*

Then $o = M_j(\text{Tuplefy}(w, \kappa))$ satisfies:

- *At any point i , $\kappa[i] \frown o[i]$ does not remove $M(w)[i]$ from Q . That is, $M(w)[i]$ is not in position $o[i]$ of $\text{Remove}(Q, \kappa[i])$, and in particular $M(w)[i] \in \text{Remove}(Q, \kappa[i] \frown o[i])$.*

Proof. The idea here is that each M_j should keep track of a single entry in an ordinal removal sequence that will remove all elements of Q except the state of the original automaton M at any one particular time. These will be the M_j in the multivariate composition, so they will be reading in both the original input (a single character a from w), and a single index from each of M_0, \dots, M_{j-1} , together forming an ordinal removal sequence $(\kappa[i])$ of length j . Each M_j then keeps track of an index, which, when added on to the end of the ordinal removal sequence does not remove the one state that must not be removed, the state of M at that point.

For reference, a picture of the situation is drawn below:



As one can see in the diagram, M_j will be reading in $w[i]$, the character that takes the automaton M from $M(w)[i]$ to $M(w)[i + 1]$, and $\kappa[i]$, the ordinal removal sequence within which its current state $w_j[i]$ is interpreted. Specifically, $w_j[i]$ will be a position in $\text{Remove}(Q, \kappa[i])$ where there isn't the current state of M , $M(w)[i]$. This will transition M_j into the state $w_j[i + 1]$, which must be a position in $\text{Remove}(Q, \kappa[i + 1])$ where there isn't the next state of M , $M(w)[i + 1]$.

Note that M_j does not get direct access to $\kappa[i+1]$, but of course it needs access to $\kappa[i+1]$ in order to determine the index for $M(w)[i+1]$ in $\text{Remove}(Q, \kappa[i+1])$ so that it can avoid it. Fortunately, if the previous automata, M_0, \dots, M_{j-1} work in canonical fashions, knowing a and $\kappa[i]$ is enough to determine $\kappa[i+1]$.

To start with, one needs to pick the starting state ($w_j[0]$) for M_j , k_0 , such that k_0 is not an index for the start state of M in $\text{Remove}(Q, \kappa[0])$. Let it be the smallest such index.

By hypothesis, the transition map induced by the character a on the automaton M , δ_a , maps states in $\text{Remove}(Q, \kappa[i])$ to states in $\text{Remove}(Q, \kappa[i+1])$. By the Permutation-Reset Lemma, one can define the transition map for M_j , δ' , with $\delta'_{(w[i], \kappa[i])}$ a permutation-reset map for any particular $w[i]$ and $\kappa[i]$ that does the avoiding required of it. \square

Finally, it's worth noting that $\kappa[i] \cap w_j[i]$, $\kappa[i+1] \cap w_j[i+1]$ satisfy the requirements on $\kappa[i]$ and $\kappa[i+1]$ in the hypothesis of this lemma. Specifically:

- The new $w_j[0]$ and $w_j[i+1]$ were chosen to avoid removing $M(w)[0]$ and $M(w)[i+1]$ from Q .
- $w_j[i+1]$ is determined by $w_j[i]$, $\kappa[i]$, and $w[i]$.
- The map δ_a maps states in $\text{Remove}(Q, \kappa[i] \cap w_j[i])$ to states in $\text{Remove}(Q, \kappa[i+1] \cap w_j[i+1])$.

This is immediate, looking at the third condition on the function the Permutation-Reset Lemma constructs.

As such the M_j in this proof are the same as the M_j in the multivariate composition for the action of M , and $\text{Tuplefy}(w_1, \dots, w_{j-1})$ is a word of ordinal removal sequences κ as above for each suitable j . As such the final character-wise map f is simply the map mapping an ordinal removal sequence \mathbf{k} of length $|Q| - 1$ to the single element of $\text{Remove}(Q, \mathbf{k})$.

Hopefully, this particular proof will shed some light on the multivariate composition used to determine $M^{\text{Trunc}}(w)$: why it is shaped the way it is shaped, and what each piece of the composition is keeping track of. Utilizing the Krohn-Rhodes Theorem, one currently has the following set of generators for the regular functions:

Theorem 4.10. *Any regular function can be written as a multivariable composition of:*

- *Truncated Permutation-Reset Transparent Moore Machine maps,*
- *Truncated Reverse Moore Machine maps,*
- *Character-wise maps,*
- *Tuplefy (allowing one to generate multiary character-wise maps),*
- *S_a for various a ,*
- *Unpad.*

It may seem like this hasn't gained much, but the next section will show that there isn't actually that much to Permutation-Reset Transparent Moore Machines. The section after will handle the reverse Moore Machine case.

5. A FURTHER BREAKING DOWN

In this section I prove that truncated permutation-reset transparent Moore Machine maps can be written as the composition of a single truncated permutation transparent Moore Machine map and a single truncated reset transparent Moore Machine map. Then I show that permutation transparent Moore Machines and reset transparent Moore Machines are actually quite familiar objects. As before, these proofs are adapted from [6], which uses vastly different notation.

Lemma 5.1. *Given a permutation-reset transparent Moore Machine $M = (\Sigma, Q, q_0, \delta)$, there is a permutation transparent Moore Machine \tilde{M} , reset transparent Moore Machine \vec{M} , and function F such that:*

$$M^{\text{Trunc}}(w) = \text{Cw}_F(\vec{M}^{\text{Trunc}}(\text{Tuplefy}(w, \tilde{M}^{\text{Trunc}}(w))), \tilde{M}^{\text{Trunc}}(w))$$

Proof. Let $\tilde{M} = (\Sigma, S_Q, id, \tilde{\delta})$ and $\vec{M} = (\Sigma \times S_Q, Q, q_0, \vec{\delta})$, where S_Q is the set of all permutations on Q , and:

If δ_a is a permutation of the states of M :

$$\tilde{\delta}(f, a) = \delta_a \circ f,$$

$$\vec{\delta}(q, (a, f)) = q.$$

If δ_a is a reset on the states of M with image $\{q_a\}$:

$$\tilde{\delta}(f, a) = f,$$

$$\vec{\delta}(q, (a, f)) = f^{-1}(q_a).$$

As desired \tilde{M} is a permutation automaton and \vec{M} is a reset automaton (notice that the identity action is necessary in case δ_a is a permutation).

Let $F : S_Q \times Q \rightarrow Q$ with $F(f, q) = f(q)$. I now claim that

$$M^{\text{Trunc}}(w) = \text{Cw}_F(\vec{M}^{\text{Trunc}}(\text{Tuplefy}(w, \tilde{M}^{\text{Trunc}}(w))), \tilde{M}^{\text{Trunc}}(w)),$$

As desired. This is easy to verify in terms of their transition relations. The intuition behind this construction is that \tilde{M} keeps track of the action of each of the permutations and \vec{M} handles resets by storing them in terms of what state one would have to start in such that after being acted on by just the permutations one winds up in the current state of M . \square

I now define a couple of transparent Moore Machines in order to refine the decomposition further.

Definition 5.2. For each n , define the *Accumulator on S_n* transparent Moore Machine AS_n :

$$AS_n = (S_n, S_n, id, \delta),$$

Where:

$$\delta_g(h) = h \cdot g,$$

Where S_n is the symmetric group on n elements with composition operation $(h \cdot g)(i) = h(g(i))$.

Definition 5.3. Define the *bit-storage* automaton:

$$Bit = (\{-, 0, 1\}, \{0, 1\}, 0, \delta),$$

Where δ_- acts as the identity, δ_0 is a reset to state 0, and δ_1 is a reset to state 1.

Since every collection of permutations can be viewed as a subset of the symmetric group S_n for some n :

Proposition 5.4. *Every truncated permutation transparent Moore Machine map M^{Trunc} can be written as:*

$$M^{\text{Trunc}}(w) = \text{Cw}_f(AS_n^{\text{Trunc}}(\text{Cw}_g(w))),$$

For some functions f and g .

What's more:

Proposition 5.5. *Every truncated reset transparent Moore Machine map M^{Trunc} can be written as:*

$$\text{Cw}_f(\text{Bit}^{\text{Trunc}}(\text{Cw}_{g_0}(w)), \dots, \text{Bit}^{\text{Trunc}}(\text{Cw}_{g_{n-1}}(w))),$$

For suitable n , f , and g_0, \dots, g_{n-1} .

Proof. Suppose $M = (\Sigma, Q, q_0, \delta)$. Choose an n such that $2^n \geq |Q|$. For every state $q \in Q$, associate a unique bitstring $b(q) \in 2^n$, where $b_k(q)$ is the bit in position k of $b(q)$, such that the start state $q_0 \in Q$ is given by the all 0s bitstring. Let $f = b^{-1}$. Suppose δ_a acts as a reset to the state q_a . Then let $g_k(a) = b_k(q_a)$. \square

As such:

Proposition 5.6. *Every truncated Moore Machine map M^{Trunc} can be written as a multi-variable composition of:*

- AS_n^{Trunc} for various n ,
- $\text{Bit}^{\text{Trunc}}$,
- Character-wise maps.

This yields a much smaller generating set for the regular functions:

Theorem 5.7. *Any regular function can be written as a multivariable composition of:*

- AS_n^{Trunc} for various n ,
- $\text{Bit}^{\text{Trunc}}$,
- Truncated Reverse Moore Machine maps,
- Character-wise maps,
- Tuplefy (allowing one to generate multiary character-wise maps),
- S_a for various a ,
- Unpad.

6. REVERSE MOORE MACHINES

Just as in previous sections, I broke down the truncated Moore Machine maps into compositions involving the accumulator on S_n , the *Bit* automaton, and character-wise maps (note that uses of Tuplefy are length-preserving, and thus actually character-wise applications of a tuple-construction map), in this section, I break down truncated reverse Moore Machine maps similarly. To save work, I will simply introduce a reversal map Rev (which is not regular) to connect truncated reverse Moore Machine maps and truncated Moore Machine Maps.

Definition 6.1. Given a word $w \in \Sigma^*$ define $\text{Rev}(w)$ to be the reversal of w .

Given a Moore Machine $M = (\Sigma, Q, q_0, \Gamma, \delta, \epsilon)$, define its reversal:

$$\text{Rev}(M) = (\Sigma, Q, q_0, \Gamma, \delta, \epsilon).$$

And similarly define the reversal of a reverse Moore Machine.

Proposition 6.2. *Given a Moore Machine M , and word w :*

$$\text{Rev}(M)(w) = \text{Rev}(M(\text{Rev}(w))).$$

What's more:

$$\text{Rev}(M)^{\text{Trunc}}(w) = \text{Rev}(M^{\text{Rest}}(\text{Rev}(w))).$$

Functions that are related in this way are said to be related by *conjugation by Rev*. This relation is reflexive and symmetric. What's more since Rev is its own inverse, if f and f' are related by conjugation and g and g' are related by conjugation, then $f \circ g$ and $f' \circ g'$ will be related by conjugation. Indeed this works for multiary functions as well:

Definition 6.3. Given an n -ary function f , say that

$$(x_0, \dots, x_{n-1}) \mapsto f(x_0, \dots, x_{n-1})$$

And

$$(x_0, \dots, x_{n-1}) \mapsto \text{Rev}(f(\text{Rev}(x_0), \dots, \text{Rev}(x_{n-1})))$$

Are related by *conjugation by Rev*.

Proposition 6.4. *Given a multi-ary composition of functions, if one replaces each function by its conjugation by Rev, the overall composition is related to the original composition by conjugation by Rev.*

Proof. It suffices to note that in the resulting composition, whenever the output of a function is fed into the input of another function, it is reversed twice, effectively doing nothing to it. \square

Additionally, conjugation by Rev does not alter character-wise functions.

It is necessary to prove that M^{Rest} can be written in terms of M^{Trunc} :

Proposition 6.5. *Given a transparent Moore Machine $M = (\Sigma, Q, q_0, \delta)$, one can write M^{Rest} as a composition of character-wise maps and M^{Trunc} .*

Proof. It is easy to verify that:

$$M^{\text{Rest}}(w) = \text{Cw}_\delta(M^{\text{Trunc}}(w), w).$$

\square

It follows from this that for any Moore Machine M , M^{Rest} can be written as a composition of character-wise maps and M^{Trunc} .

Proposition 6.6. *Any truncated reverse Moore Machine map can be written as the multi-variate composition of:*

- RAS_n^{Trunc} , where RAS_n is the reversal of AS_n , for various n ,
- $RBit^{\text{Trunc}}$, where $RBit$ is the reversal of Bit ,
- Character-wise maps.

Proof. Every reverse Moore Machine is $\text{Rev}(M)$ for some M . As such, one can write $\text{Rev}(M)^{\text{Trunc}}$ as:

$$\text{Rev} \circ M^{\text{Rest}} \circ \text{Rev}.$$

One can write M^{Rest} as a multiary composition of AS_n^{Trunc} for various n , Bit^{Trunc} , and character-wise maps, so by the conjugation of compositions lemma, one can write $\text{Rev}(M)^{\text{Trunc}}$ as a multiary composition of the conjugations of those components. \square

Also note:

Proposition 6.7. *Given a Moore Machine M , one can write $M(w)$ as the multivariable composition of a truncated Moore Machine map and $S_{\#}$.*

Proof. Augment M to M' by allowing it to interpret the input $\#$ (it may do so in any way it likes). Then:

$$M(w) = M^{\text{Trunc}}(S_{\#}(w)).$$

□

As one final refinement of the generating set, I show that RAS_n is unnecessary as a generator.

7. REMOVING THE REVERSE ACCUMULATOR

Note that AS_n and RAS_n are very similar automata. For AS_n , one interprets the input character $w[i]$ as a permutation relating $AS_n(w)[i]$ and $AS_n(w)[i + 1]$. For RAS_n , one interprets the input character $w[i]$ as a permutation relating $RAS_n(w)[i + 1]$ and $RAS_n(w)[i]$. Since every permutation has an inverse, shouldn't these two automata be the same up to a suitable character-wise map on the inputs? Alas, the distinction is more subtle: there is another constraint on the runs of AS_n and RAS_n . For AS_n , the first character of its run is specified to be id . For RAS_n , the last character of its run is specified to be id .

Compare $AS_n(w)$ and $RAS_n(\text{Cw}_{\text{inverse}}(w))$ on some generic five character input $w = abcde$:

w	a	b	c	d	e	
$AS_n(w)$	id	a	ab	abc	$abcd$	$abcde$
$RAS_n(\text{Cw}_{\text{inverse}}(w))$	$(abcde)^{-1}$	$(bcde)^{-1}$	$(cde)^{-1}$	$(de)^{-1}$	e^{-1}	id

In addition to applying a suitable transformation to the inputs of AS_n , one must also apply a suitable transformation to the outputs of AS_n if one wants to produce the output of RAS_n . Specifically, if one multiplies every character in the output of AS_n on the left by the inverse of the last character of the output, it will ensure that the new last character of the output is id , but still maintain the transition relationships. This requires passing the information of the last character of AS_n to every other position.

First, I must introduce the *mask* of the word w , a word $\text{Mask}(w)$ which is all 0s up to the length of w , followed by a 1. This is computed simply by taking $S_1(\text{Cw}_0(w))$ where 0 is the constant 0 map. Despite its simplicity, this word will be key to performing the computation.

Consider the reverse reset Transparent Moore Machine $R = (\{0, 1\} \times S_n, S_n, id, \delta)$ with:

$$\delta(q, a) = \begin{cases} b & a = (1, b) \\ q & a = (0, b) \end{cases}$$

This is a reverse reset transparent Moore Machine, and so R^{Trunc} can be written in terms of character-wise maps and $R\text{Bit}^{\text{Trunc}}$.

Consider the action of R^{Trunc} on $\text{Cw}_{\text{pair}}(\text{Mask}(w), AS_n(w))$:

w	a	b	c	d	e	
$\text{Mask}(w)$	0	0	0	0	0	1
$AS_n(w)$	id	a	ab	abc	$abcd$	$abcde$
$R^{\text{Trunc}}(\text{Cw}_{\text{pair}}("))$	$abcde$	$abcde$	$abcde$	$abcde$	$abcde$	$abcde$

I now have a word which can be combined with $AS_n(w)$ via the appropriate bitwise map (multiplying by the inverse on the left) to produce $RAS_n(\text{Cw}_{\text{inverse}}(w))$.

However, what I wanted was $RAS_n^{\text{Trunc}}(\text{Cw}_{\text{inverse}}(w))$. One can attain that by combining $RAS_n(\text{Cw}_{\text{inverse}}(w))$ with $\text{Mask}(w)$ bitwise to replace the last character of $RAS_n(\text{Cw}_{\text{inverse}}(w))$ with a $\#$ and then using Unpad to remove it.

Letting f map pairs of the form $(0, a)$ to a and pairs of the form $(1, a)$ to $\#$:

w	a	b	c	d	e	
$\text{Mask}(w)$	0	0	0	0	0	1
$RAS_n(\text{Cw}_{\text{inverse}}(w))$	$(abcde)^{-1}$	$(bcde)^{-1}$	$(cde)^{-1}$	$(de)^{-1}$	e^{-1}	id
$\text{Cw}_f(")$	$(abcde)^{-1}$	$(bcde)^{-1}$	$(cde)^{-1}$	$(de)^{-1}$	e^{-1}	$\#$
$\text{Unpad}(")$	$(abcde)^{-1}$	$(bcde)^{-1}$	$(cde)^{-1}$	$(de)^{-1}$	e^{-1}	

This composition produces the desired output and works in general. Thus, the final form of my theorem:

Theorem 7.1. *Any regular function can be written as a multivariable composition of:*

- AS_n^{Trunc} for various n ,
- Bit^{Trunc} ,
- $RBit^{\text{Trunc}}$,
- Character-wise maps,
- Tuplefy,
- S_a for various a ,
- Unpad .

8. FURTHER RESEARCH

While of independent interest, these decomposition results have significantly streamlined proofs involving nonstandard models of the Weak Second order Theory of One Successor (WS1S). I intend to publish my findings in two papers, one presenting a complete axiomatization of WS1S, and the other presenting some results towards a classification of the nonstandard models of WS1S.

While thinking about automata as transducers instead of acceptors takes one away from the underlying logic, it takes one closer to real-world applications of automata. One is then lead to ask similar questions about functions whose graphs are recognized by Büchi automata, Tree automata, and Rabin automata. The determinization-harvester decomposition can be adapted to trees, but is there an analog to the Krohn-Rhodes theorem for trees in this context? What about in the case of Büchi automata or Rabin automata, for which there is no end of the input to start the harvester running backwards from? Can nice generators still be found?

There is still much work to be done in establishing a Büchi-Elgot-Trakhtenbrot theorem for graphs. There are several nice candidates for a monadic second-order logic of graphs, and some nice notions of automata operating on graphs, but no full correspondence between

them. The current state of the art is Courcelle's Theorem (see, e.g. [2]), which allows one to translate questions in a graph logic to tree automata operating on a tree decomposition of the original graph, but not back. Fortunately, this is the direction of most interest to applications. But perhaps an approach which instead of trying to connect formulas and acceptors, connected describable functions and transducers, would shed light on the problem?

Finally, also note that it is not possible to break down the AS_n generators much further. Of course for large enough n one may decompose the symmetric group S_n as a semidirect product of the alternating group A_n and S_2 , and use this decomposition to guide a slight decomposition of AS_n , but this doesn't gain anything. In light of the simplicity of the (large) alternating groups, it is likely no further decomposition in the function composition context is possible. One would then like a proof that, for instance, accumulators on the cyclic groups do not suffice, in a way that hopefully sheds some light on what behavior symmetric groups capture that cyclic groups cannot. Alternately, a decomposition of the accumulators on the symmetric groups in terms of accumulators on the cyclic groups would be a remarkable result.

ACKNOWLEDGMENT

The author wishes to acknowledge fruitful discussions with Scott Messick. This work would not have been possible without encouragement and guidance from my Ph.D. advisor, Anil Nerode.

REFERENCES

- [1] Benedikt, M., Libkin, L., Schwentick, T., & Segoufin, L. (2003). Definable relations and first-order query languages over strings. *Journal of the ACM*, 50(5), 694-751.
- [2] Downey, R. G., & Fellows, M. R. (1999). *Parameterized complexity*. New York: Springer.
- [3] Egri-Nagy, A., & Nehaniv, C. (2005). Algebraic Hierarchical Decomposition of Finite State Automata: Comparison of Implementations for Krohn-Rhodes Theory. *Implementation and Application of Automata Lecture Notes in Computer Science*, 315-316.
- [4] Egri-Nagy, A., & Nehaniv, C. (2006). Making Sense of the Sensory Data Coordinate Systems by Hierarchical Decomposition. *Lecture Notes in Computer Science Knowledge-Based Intelligent Information and Engineering Systems*, 333-340.
- [5] Eilenberg, S., & Tilson, B. (1976). *Automata, Languages, and Machines: Volume B*. New York: Academic Press.
- [6] Ginzburg, A. (1968). *Algebraic theory of automata*. New York: Academic Press.
- [7] Krohn, K., & Rhodes, J. (1965). Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines *Transactions of the American Mathematical Society*, 450-450.

APPENDIX A. LIST OF NOTATION

M, N	Moore machines
R, P	Reverse Moore machines
A	Finite automata
Σ, Γ	Finite alphabets
Σ^*	The set of finite words with characters from Σ
Q	Finite set of states
q_0	Initial state
q_f	Final state
I	Set of initial states
F	Set of final states
δ	Transition relation $Q \times \Sigma \rightarrow Q$
$\delta_a(q)$	$= \delta(q, a)$
$ w $	The length of the word w
$w[i]$	The character of word w in position i (first character is $w[0]$, last is $w[w - 1]$)
$w_0 \widehat{\ } w_1$	Concatenation
$S_a(w)$	$= w \widehat{\ } a$
Trunc	Remove the last character of a word
Rest	Remove the first character of a word
$M(w)$	The output of Moore machine M on input w
$M^{\text{Trunc}}(w)$	$= \text{Trunc}(M(w))$
$M^{\text{Rest}}(w)$	$= \text{Rest}(M(w))$
$\#$	Dummy character for padding ends of words
Tuplefy	Combine inputs words in parallel, padding with $\#$
Cw_f	Character-wise application of function f
$\det(A)$	Determinization of automaton A
$\text{harv}(A)$	Harvester construction for automaton A
$\text{Pair}(x, y)$	$= (x, y)$
Unpad	Removes terminal $\#$ characters
\overline{M}	Moore machine which keeps track of a state that Moore machine M is not in
\widehat{M}	Moore machine which keeps track of the rest of the information about the state of M
$\mathcal{O}_{k,n}$	The set of ordinal removal sequences for n of length k
\mathbf{k}	$= (k_0, \dots, k_{i-1})$
$\text{Remove}(L, \mathbf{k})$	Applies ordinal removal sequence \mathbf{k} to ordered collection L
\vec{M}	A reset Moore Machine
\tilde{M}	A permutation Moore Machine
AS_n	The accumulator on S_n automaton
Bit	The bit storage automaton
RAS_n	The reverse accumulator on S_n
$RBit$	The reverse bit storage automaton
$\text{Rev}(w)$	The reverse of word w